

*lib*_{CH}

C++-Bibliothek
von
Prof. Dr. Christian Heinlein

27. April 2021

1 Einleitung

Die hier beschriebene Bibliothek befindet sich in Dateien mit der Endung `.ch`, und ihr gesamter Code befindet sich im Namensraum `CH`.

Da es für Makros keine Namensräume gibt, beginnen alle Makronamen vorsichtshalber mit dem Präfix `CH_`, z. B. `CH_TYPE`. Wenn beim Einbinden einer Definitionsdatei das Makro `USING_CH` definiert ist, werden aber zusätzlich „Aliase“ ohne dieses Präfix definiert, z. B. `TYPE`. Bei der nachfolgenden Beschreibung werden die Präfixe `CH::` und `CH_` der Einfachheit halber weggelassen.

Damit durch `using namespace CH` in einem Anwendungsprogramm keine internen Namen des Namensraums `CH` sichtbar werden (die dann zu Konflikten führen können), beginnen interne Namen ebenfalls mit dem Präfix `CH_`. (Die Verlagerung interner Namen in einen untergeordneten Namensraum würde dieses Problem nicht vollständig lösen, weil dann zumindest der Name dieses Unternamensraums sichtbar wäre.)

Die Implementierung verwendet zahlreiche Sprachmittel von C++11, wie z. B. range-based for loops, variadic templates, perfect forwarding usw., sowie einige von C++14 und C++17, z. B. fold expressions und deduction guides. Deshalb braucht man einen Compiler, der C++17 ausreichend fehlerfrei unterstützt, z. B. `g++` ab Version 9 oder `clang++` ab Version 7.

`g++` Version 7 und 8 akzeptieren deduction guides mit leerer Parameterliste nicht. `clang++` Version 5 hat möglicherweise ein Problem mit variadic deduction guides. Deshalb funktionieren diese Compiler-Versionen nicht.

`clang++` Version 6 wurde nicht getestet.

Der Code wurde konkret getestet mit:

- `g++` 7.5.0 (funktioniert nicht)
- `g++` 8.2.1 (funktioniert nicht)
- `g++` 9.3.1 (funktioniert)
- `g++` 10.2.1 (funktioniert)
- `clang++` 5.0.2. (funktioniert nicht)
- `clang++` 7.0.1. (funktioniert)
- `msvc` 19.28 auf `godbolt.org` (funktioniert)

2 Hilfsmittel für Template-Metaprogrammierung (meta.ch)

2.1 Beschreibung

Siehe Kommentare in der Implementierung.

2.2 Implementierung

```
// Copyright (C) 2021 Prof. Dr. Christian Heinlein
```

```
#ifndef CH_META_CH
#define CH_META_CH 2021'03'24
```

```
#include <type_traits>
```

```
namespace CH {
```

2.2.1 Meta-Zähler

```
// Size<i> ist Untertyp von Size<i-1> und hat Größe i.
```

```
template <int i>
struct CH_Size : CH_Size<i-1> {
    char x;
};
```

```
// Size<1> hat Größe 1.
```

```
template <>
struct CH_Size<1> {
    char x;
};
```

```
// Prinzipiell beliebiger Maximalwert eines Zählers.
```

```
const int CH_N = 64;
```

```
// Zähler und Hilfszähler mit dem Namen name definieren
```

```
// und mit Wert 1 initialisieren.
```

```
// Wenn bereits ein Zähler mit diesem Namen (im selben Namensraum)
```

```
// definiert wurde, ist die Anweisung wirkungslos (und insbesondere
```

```
// nicht fehlerhaft).
```

```
#define CH_DEF_CNT(name) \
    CH::CH_Size<1> name##_1 (CH::CH_Size<1>); \
    CH::CH_Size<1> name##_2 (CH::CH_Size<1>); \
    CH::CH_Size<1> name##_3 (CH::CH_Size<1>); \
    CH::CH_Size<1> name##_4 (CH::CH_Size<1>); \
    CH::CH_Size<1> name##_5 (CH::CH_Size<1>);
```

```
// Zähler und Hilfszähler mit dem Namen name abfragen.
```

```
#define CH_GET_CNT(name) \
    CH_GET_CNT1(name)
#define CH_GET_CNT1(name) \
    sizeof name##_1(CH::CH_Size<CH::CH_N * CH_GET_CNT2(name)>())
#define CH_GET_CNT2(name) \
```

```

        sizeof name##_2(CH::CH_Size<CH::CH_N * CH_GET_CNT3(name)>())
#define CH_GET_CNT3(name) \
        sizeof name##_3(CH::CH_Size<CH::CH_N * CH_GET_CNT4(name)>())
#define CH_GET_CNT4(name) \
        sizeof name##_4(CH::CH_Size<CH::CH_N * CH_GET_CNT5(name)>())
#define CH_GET_CNT5(name) \
        sizeof name##_5(CH::CH_Size<CH::CH_N>())

// Zähler mit dem Namen name um 1 erhöhen.
// Wenn der neue Wert ein Vielfaches von N bzw. N*N bzw. ... ist,
// wird auch der Hilfszähler mit der Nummer 2 bzw. 3 bzw. ...
// um 1 erhöht.
// Wenn man die Reihenfolge der Deklarationen umkehren würde,
// würden die Hilfszähler jeweils einen Schritt später erhöht werden,
// was aber genauso funktionieren würde.
#define CH_INC_CNT(name) \
    CH::CH_Size<CH_GET_CNT(name)+1> name##_1 \
        (CH::CH_Size<CH_GET_CNT(name)+1>); \
    CH::CH_Size<CH_GET_CNT(name)/CH::CH_N+1> name##_2 \
        (CH::CH_Size<CH_GET_CNT(name)/CH::CH_N+1>); \
    CH::CH_Size<CH_GET_CNT(name)/CH::CH_N/CH::CH_N+1> name##_3 \
        (CH::CH_Size<CH_GET_CNT(name)/CH::CH_N/CH::CH_N+1>); \
    CH::CH_Size<CH_GET_CNT(name)/CH::CH_N/CH::CH_N/CH::CH_N+1> name##_4 \
        (CH::CH_Size<CH_GET_CNT(name)/CH::CH_N/CH::CH_N/CH::CH_N+1>); \
    CH::CH_Size<CH_GET_CNT(name)/CH::CH_N/CH::CH_N/CH::CH_N/CH::CH_N+1> name##_5 \
        (CH::CH_Size<CH_GET_CNT(name)/CH::CH_N/CH::CH_N/CH::CH_N/CH::CH_N+1>);

#ifdef USING_CH
#define DEF_CNT CH_DEF_CNT
#define GET_CNT CH_GET_CNT
#define INC_CNT CH_INC_CNT
#endif

```

2.2.2 Sonstiges

```

// Hilfstyp für CH_IF.
template <bool ... B>
struct conjunction {
    static const int value = (B && ...);
};

// IF(x) kann am Ende einer Schablonenparameterliste verwendet werden,
// damit die zugehörige Schablone nur verwendet werden kann, wenn die
// Bedingungen ... zur Übersetzungszeit erfüllt sind.
// Der Typ int und der Wert 0 sind willkürlich, aber sie müssen
// zusammenpassen.
#define CH_IF(...) \
    std::enable_if_t<CH::conjunction<__VA_ARGS__>::value, int> = 0

```

```

// SAME(X, Y) und CONV(X, Y) kann innerhalb von IF(.....) verwendet
// werden, um auszudrücken, dass die Typen X und Y gleich sein müssen
// bzw. dass X implizit in Y umwandelbar sein muss.
#define CH_SAME(X, Y) std::is_same_v<X, Y>
#define CH_CONV(X, Y) std::is_convertible_v<X, Y>

#ifdef USING_CH
#define IF CH_IF
#define SAME CH_SAME
#define CONV CH_CONV
#endif

}

#endif

```

3 Weitere Hilfsmittel (ut i l . ch)

3.1 Beschreibung

3.1.1 Nilwerte

Die Konstante `nil` repräsentiert einen generischen Nilwert, der prinzipiell in jeden Typ `T` umgewandelt werden kann. Diese Umwandlung liefert den Wert `nil_v<T>`, der standardmäßig gleich `T()` ist. Wenn es für einen Typ `T` keinen parameterlosen Konstruktor gibt oder dieser wider Erwarten nicht den gewünschten Wert liefert, kann die Variablenschablone `nil_v` geeignet spezialisiert werden.

3.1.2 bool-artige Werte

Der Typ `boollike` ist ein Synonym für einen elementaren Typ wie z. B. `void*`, der implizit nach `bool` umgewandelt werden kann und dessen Werte deshalb wie `bool`-Werte verwendet werden können. Andererseits kann dieser Typ in (möglichst) keinen anderen Typ umgewandelt werden, und er besitzt (möglichst) keine anwendbaren Operationen (wie z. B. arithmetische Operatoren).

Wenn ein benutzerdefinierter Typ eine implizite Umwandlung nach `boollike` statt nach `bool` besitzt, können seine Objekte zwar wie `bool`-Werte (z. B. in Schleifenbedingungen) verwendet werden, aber nicht als numerische Werte, was oft zu unerwünschten Mehrdeutigkeiten führen würde.

Für einen `bool`-Wert `b` liefert `boollikeval(b)` den zu `b` korrespondierenden Wert des Typs `boollike`.

3.1.3 Dreiwertige Logik

Der Typ `bool3` realisiert eine dreiwertige Logik mit den Wahrheitswerten `false` (sicher nicht wahr), `maybe` (eventuell wahr) und `true` (sicher wahr), wobei `false` kleiner als `maybe` und `maybe` kleiner als `true` ist.

`bool3` ist kompatibel mit `bool`, d. h. die `bool`-Werte `false` und `true` können auch als `bool3`-Werte verwendet werden. Umgekehrt kann ein `bool3`-Wert auch als `bool`-Wert verwendet werden, der genau dann `true` ist, wenn der `bool3`-Wert `true` ist, d. h. `false` und `maybe` werden beide auf den `bool`-Wert `false` abgebildet.

Die Und- bzw. Oder-Verknüpfung (Operatoren `&` und `&&` bzw. `|` und `||`) zweier Werte des Typs `bool3` (oder auch `bool`) liefert das Minimum bzw. Maximum der beiden Werte.

Die Negation (Operatoren `~` und `!`) von `true` bzw. `false` ist wie gewohnt `false` bzw. `true`, die Negation von `maybe` ist `maybe`.

3.1.4 Unveränderte Weitergabe von Makroargumenten

Wenn ein Text `xyz` Kommas enthält (z. B. `std::pair<int, int>`), wird er bei der Übergabe an ein Makro in mehrere Makroargumente zerlegt. Um dies zu verhindern, kann er mit dem Makro `LIT` geklammert werden: `LIT(xyz)`.

3.2 Implementierung

```
// Copyright (C) 2021 Prof. Dr. Christian Heinlein

#ifndef CH_UTIL_CH
#define CH_UTIL_CH 2021'04'27

namespace CH {

// Nil-Wert des Typs T.
template <typename T>
const T nil_v = T();

// Generischer Nilwert,
// der implizit in jeden Typ T umgewandelt werden kann.
const struct {
    template <typename T>
    operator T () const {
        return nil_v<T>;
    }
} nil;

// bool-artiger Typ.
using boollike = void*;

// Umwandlung von bool nach boollike.
inline boollike boollikeval (bool b) {
    static char dummy;
    return b ? &dummy : nullptr;
}

// Dreiwertige Logik.
struct bool3 {
    // 0/1/2 bedeutet false/maybe/true.
    int value;

    // Interner Konstruktor.
    explicit bool3 (int value) : value(value) {}
};
};
```

```

    // Implizite Umwandlung von und nach bool.
    bool3 (bool value = false) : value(2 * value) {}
    operator bool () const { return value == 2; }
};

// Konstante maybe.
const bool3 maybe(1);

// Vergleichsoperatoren.
// Damit Anwendungen mit "gemischten" Operanden (ein Operand hat Typ
// bool3, der andere einen Typ, der in bool umgewandelt werden kann)
// eindeutig sind, sind pro Operator drei Definitionen nötig.
#define AUX(op) \
    inline bool3 operator op (bool3 x, bool3 y) { return x.value op y.value; } \
    inline bool3 operator op (bool3 x, bool y) { return x op bool3(y); } \
    inline bool3 operator op (bool x, bool3 y) { return bool3(x) op y; }
AUX(<)
AUX(<=)
AUX(==)
AUX(!=)
AUX(>=)
AUX(>)
#undef AUX

// Negation.
inline bool3 operator~ (bool3 x) { return bool3(2 - x.value); }
inline bool3 operator! (bool3 x) { return bool3(2 - x.value); }

// Konjunktion und Disjunktion.
inline bool3 operator&= (bool3& x, bool3 y) {
    if (y.value < x.value) x.value = y.value;
    return x;
}
inline bool3 operator|= (bool3& x, bool3 y) {
    if (y.value > x.value) x.value = y.value;
    return x;
}
#define AUX(op, opeq) \
    inline bool3 operator op (bool3 x, bool3 y) { return x opeq y; } \
    inline bool3 operator op (bool3 x, bool y) { return x op bool3(y); } \
    inline bool3 operator op (bool x, bool3 y) { return bool3(x) op y; }
AUX(&, &=)
AUX(|, |=)
AUX(&&, &&=)
AUX(||, ||=)
#undef AUX

// Hilfsmakro zur Klammerung von Text, der Kommas enthalten kann,
// aber trotzdem als ein einziges Makroargument übergeben werden soll.
#define CH_LIT(...) __VA_ARGS__

```

```
#ifdef USING_CH
#define LIT CH_LIT
#endif

}

#endif
```

4 Ablaufverfolgung

4.1 Beschreibung

Die Klasse `Trace` ermöglicht eine einfache Ablaufverfolgung, die zur Übersetzungszeit durch Definition des Makros `CH_TRACE` (z. B. mittels `-D CH_TRACE`) für ein Programm oder eine Quelldatei aktiviert werden kann.

Die Klasse verwaltet eine aktuelle Einrückung, die anfangs leer ist und für jede Ebene um 2 Leerzeichen erhöht wird.

Der Konstruktor der Klasse kann mit beliebig vielen beliebigen Parametern aufgerufen werden. Er gibt eine Zeile mit der aktuellen Einrückung aus, die aus einer öffnenden geschweiften Klammer und den Werten der Parameter, jeweils getrennt durch ein Leerzeichen, besteht. (Für die Typen der Parameter muss es hierfür geeignete Definitionen des Ausgabeoperators `<<` geben.) Anschließend wird die aktuelle Einrückung um eine Ebene erhöht.

Der Destruktor der Klasse (der üblicherweise implizit aufgerufen wird) reduziert die aktuelle Einrückung um eine Ebene und gibt dann eine Zeile mit der neuen aktuellen Einrückung aus, die aus einer schließenden geschweiften Klammer besteht.

Der Klammeroperator der Klasse kann ebenfalls mit beliebig vielen beliebigen Parametern aufgerufen werden. Er gibt eine Zeile mit der aktuellen Einrückung aus, die aus den Werten der Parameter, jeweils getrennt durch ein Leerzeichen, besteht.

Alle Ausgaben gehen auf `std::clog`, das genauso wie `std::cerr` dem `stderr` von C und damit Filedeskriptor 2 entspricht.

Das Makro `TRACE_BLOCK` kann mit beliebig vielen beliebigen Parametern aufgerufen werden. Es deklariert eine Variable des Typs `Trace` mit einem internen Namen und gibt seine Parameter an den aufgerufenen Konstruktor weiter.

Das Makro `TRACE_LINE` kann ebenfalls mit beliebig vielen beliebigen Parametern aufgerufen werden. Es ruft den Klammeroperator auf der zuvor mittels `TRACE_BLOCK` im selben Anweisungsblock definierten Variablen auf und gibt seine Parameter an ihn weiter.

Diese beiden Makros werden allerdings nur wie beschrieben definiert, wenn beim Einbinden von `trace.ch` das Makro `CH_TRACE` definiert ist. Andernfalls ist die Definition beider Makros leer und ihre Verwendung damit wirkungslos.

Eine typische Verwendung sieht daher wie folgt aus:

```
#include <iostream>
#include "trace.ch"

void countevenodd (int a [], int n) {
    CH_TRACE_BLOCK("countevenodd", n)
    int even = 0, odd = 0;
    for (int i = 0; i < n; i++) {
        CH_TRACE_BLOCK("for", i, a[i])
        if (a[i] % 2 == 0) {
            even++;
            CH_TRACE_LINE("even", even);
        }
        else {
            odd++;
            CH_TRACE_LINE("odd", odd);
        }
    }
    std::cout << even << " " << odd << std::endl;
}
```

4.2 Implementierung

```
#ifndef CH_TRACE_CH
#define CH_TRACE_CH 2021'04'27

#include <iostream>
#include <string>

namespace CH {

struct Trace {
    static std::string indent;

    std::string ind;

    template <typename ... TT>
    Trace (TT ... xx) : ind(indent) {
        std::clog << indent << "{";
        ((std::clog << " " << xx), ...);
        std::clog << std::endl;
        indent += " ";
    }

    template <typename T, typename ... TT>
    void operator() (T x, TT ... xx) {
        std::clog << indent << x;
        ((std::clog << " " << xx), ...);
        std::clog << std::endl;
    }
}
```



```

    ~Trace () {
        indent = ind;
        std::clog << indent << "}" << std::endl;
    }
};

inline std::string Trace::indent = "";

#ifdef CH_TRACE

#define CH_TRACE_BLOCK(...) \
    CH::Trace __trace(__VA_ARGS__);

#define CH_TRACE_LINE(...) \
    __trace(__VA_ARGS__);

#else

#define CH_TRACE_BLOCK(...) /* leer */
#define CH_TRACE_LINE(...) /* leer */

#endif

#ifdef USING_CH
#define TRACE_BLOCK CH_TRACE_BLOCK
#define TRACE_LINE CH_TRACE_LINE
#endif
}

#endif

```

5 Generische Funktionsobjekte (func.ch)

5.1 Beschreibung

Siehe Kommentare in der Implementierung.

5.2 Implementierung

```

// Copyright (C) 2021 Prof. Dr. Christian Heinlein

#ifndef CH_FUNC_CH
#define CH_FUNC_CH 2021'03'22

#include <utility>

namespace CH {

// Basistyp aller im folgenden definierten Typen F, der die Operatoren
// Negation (!), Konjunktion (&&), Disjunktion (||) und Komposition (,)
// für alle Funktionsobjekte definiert, sodass ein Ausdruck wie z. B.
// (rem(2), eq(0)) && (rem(4), !eq(0)) ein Funktionsobjekt konstruiert,

```

```

// das die Funktion  $f(x) = x^2 == 0 \ \&\& \ !(x^4 == 0)$  implementiert.
// Die Funktionsschablonen neg, conj, disj und comp werden später
// definiert.
template <typename F>
struct CH_common_t {
    // Das aktuelle Objekt mit seinem eigentlichen Typ F.
    F& self () const {
        return *(F*)this;
    }

    // Negation.
    auto operator! () const {
        return neg(self());
    }

    // Konjunktion.
    template <typename F2>
    auto operator&& (const F2& f2) const {
        return conj(self(), f2);
    }

    // Disjunktion.
    template <typename F2>
    auto operator|| (const F2& f2) const {
        return disj(self(), f2);
    }

    // Komposition.
    template <typename F2>
    auto operator, (const F2& f2) const {
        return comp(f2, self());
    }
};

// Die Funktionsschablone neg kann mit einem Funktionsobjekt f eines
// beliebigen Typs F aufgerufen werden und liefert als Resultat ein
// Objekt des Hilfstyps neg_t<F>, das das Objekt f enthält.
// Die Elementfunktion operator() dieses Typs kann mit beliebigen
// Werten xx ... aufgerufen werden und liefert als Resultat den Wert
// !f(std::forward<XX>(xx) ...).

template <typename F>
struct neg_t : CH_common_t<neg_t<F>> {
    F f;
    neg_t (const F& f) : f(f) {}

    template <typename ... XX>
    auto operator() (XX&& ... xx) const {
        return !f(std::forward<XX>(xx) ...);
    }
};

```

```

template <typename F>
auto neg (const F& f) {
    return neg_t<F>(f);
}

// Für einen Namen name (z. B. conj) und einen zugehörigen Ausdruck expr
// (z. B. f1(std::forward<XX>(xx) ...) && f2(std::forward<XX>(xx) ...))
// definiert AUX(name, expr) eine Funktionsschablone name (z. B. conj),
// die mit zwei Funktionsobjekten f1 und f2 mit beliebigen Typen F1 und
// F2 aufgerufen werden kann und als Resultat ein Objekt des Hilfstyps
// name_t (z. B. conj_t) liefert, das die Objekte f1 und f2
// enthält.
// Die Elementfunktion operator() dieses Typs kann mit beliebigen Werten
// xx ... aufgerufen werden und liefert als Resultat den Wert expr.
#define AUX(name, expr) \
    template <typename F1, typename F2> \
    struct name##_t : CH_common_t<name##_t<F1, F2>> { \
        F1 f1; \
        F2 f2; \
        name##_t (const F1& f1, const F2& f2) : f1(f1), f2(f2) {} \
        \
        template <typename ... XX> \
        auto operator() (XX&& ... xx) const { \
            return expr; \
        } \
    }; \
    template <typename F1, typename F2> \
    auto name (const F1& f1, const F2& f2) { \
        return name##_t<F1, F2>(f1, f2); \
    }

// Verwendung von AUX für Konjunktion, Disjunktion und Komposition.
AUX(conj, f1(std::forward<XX>(xx) ...) && f2(std::forward<XX>(xx) ...))
AUX(disj, f1(std::forward<XX>(xx) ...) || f2(std::forward<XX>(xx) ...))
AUX(comp, f1(f2(std::forward<XX>(xx) ...)))

#undef AUX

// Für einen Namen name (z. B. conj) definiert AUX(name) eine
// variadische Funktion name (z. B. conj) als Verallgemeinerung der
// gleichnamigen zweistelligen Funktion, die mit zwei oder mehr
// Funktionsobjekten f1, f2, ff ... aufgerufen werden kann.
// Da bei einem Aufruf mit genau zwei Funktionsobjekten f1, f2 die
// nicht-variadische zweistellige Funktion bevorzugt wird, wird die
// hier definierte variadische Funktion tatsächlich immer mit mindestens
// drei Funktionsobjekten aufgerufen, d. h. das Bündel ff ist niemals
// leer.
#define AUX(name) \
    template <typename F1, typename F2, typename ... FF> \
    auto name (const F1& f1, const F2& f2, const FF& ... ff) { \
        return name(name(f1, f2), ff ...); \
    }

```

```

// Verwendung von AUX für Konjunktion, Disjunktion und Komposition.
AUX(conj)
AUX(disj)
AUX(comp)

#undef AUX

// Für einen Namen name (z. B. eq) und ein zugehöriges Operatorsymbol
// oper (z. B. ==) definiert AUX(name, oper) eine Funktionsschablone
// name (z. B. eq), die mit einem Wert y eines beliebigen Typs Y
// aufgerufen werden kann und als Resultat ein Objekt des Hilfstyps
// name_t<Y> (z. B. eq_t<Y>) liefert, das den Wert y enthält.
// Die Elementfunktion operator() dieses Typs kann mit einem Wert x
// eines beliebigen Typs X aufgerufen werden und liefert als Resultat
// den Wert x oper y (z. B. x == y).
#define AUX(name, oper) \
    template <typename Y> \
    struct name##_t : CH_common_t<name##_t<Y>> { \
        Y y; \
        name##_t (const Y& y) : y(y) {} \
        \
        template <typename X> \
        auto operator() (const X& x) const { \
            return x oper y; \
        } \
    }; \
    \
    template <typename Y> \
    name##_t<Y> name (const Y& y) { \
        return name##_t<Y>(y); \
    }

// Verwendung von AUX für Vergleichsoperatoren.
AUX(eq, ==)
AUX(ne, !=)
AUX(gt, >)
AUX(ge, >=)
AUX(lt, <)
AUX(le, <=)

// Verwendung von AUX für arithmetische Operatoren.
AUX(add, +)
AUX(sub, -)
AUX(mul, *)
AUX(div, /)
AUX(rem, %)

#undef AUX

}

#endif

```

6 Sequenzen (**seq.ch**)

6.1 Beschreibung

Für einen Elementtyp `T` stellt ein Objekt des Typs `seq<T>` eine Sequenz von Objekten des Typs `T` dar.

Als Sonderfall stellt ein Objekt des Typs `seq<void>` immer eine „generische“ leere Sequenz dar, die implizit in jeden anderen Typ `seq<T>` umgewandelt werden kann.

Im folgenden bezeichnet `s` immer eine Sequenz eines solchen Typs `seq<T>`.

6.1.1 Konstruktoren und einfache Abfragen

Der Typ `seq<T>` besitzt folgende Konstruktoren:

- `seq<T>()` liefert eine leere Sequenz.
Dieser Konstruktor kann auch implizit aufgerufen werden.
Wenn man den Typparameter `T` weglässt, erhält man eine generische leere Sequenz mit Typ `seq<void>`.
- Für einen Wert `x` mit Typ `seq<void>` liefert `seq<T>(x)` ebenfalls eine leere Sequenz.
Dieser Konstruktor kann auch implizit aufgerufen werden.
- Für einen oder mehrere Werte `x` mit Typ `T` (bzw. beliebigen Typen, die jeweils implizit in `T` umgewandelt werden können) liefert `seq<T>(x ...)` eine Sequenz mit den Elementen `x`.
Dieser Konstruktor muss explizit aufgerufen werden.
Wenn man den Typparameter `T` weglässt, müssen die Werte `x` einen „gemeinsamen“ Typ gemäß `std::common_type` besitzen, der dann als Elementtyp der Sequenz verwendet wird.
- Für eine Initialisiererliste `xs` mit Typ `std::initializer_list<T>` liefert `seq<T>(xs)` eine Sequenz mit den Elementen von `xs`.
Dieser Konstruktor kann auch implizit aufgerufen werden.
- Für `xs` mit einem beliebigen anderen Typ, der nicht implizit in `T` umgewandelt werden kann, liefert `seq<T>(xs)` eine Sequenz mit den Elementen `x`, die sich aus der Anweisung

```
for (auto x : xs) .....
```

ergeben. Falls diese Anweisung nicht korrekt ist, führt die Verwendung dieses Konstruktors zu einem Übersetzungsfehler.

Insbesondere kann `xs` ein beliebiger STL-Container sein.

Dieser Konstruktor muss explizit aufgerufen werden. Aus diesem Grund gibt es zusätzlich den Konstruktor für eine Initialisiererliste, obwohl er nur ein Spezialfall dieses Konstruktors ist.

- Für Iteratoren `b` und `e` eines beliebigen Typs, der nicht implizit in `T` umgewandelt werden kann, liefert `seq<T>(b, e)` eine Sequenz mit den Elementen `x`, die sich aus der Anweisung

```
for (auto i = b; i != e; i++) {  
    auto x = *i;  
    .....  
}
```

ergeben. Falls diese Anweisung nicht korrekt ist, führt die Verwendung dieses Konstruktors zu einem Übersetzungsfehler.

Insbesondere können `b` und `e` beliebige STL-Iteratoren sein.

Für eine Sequenz `s` liefert `*s` die Länge von `s`, d. h. die Anzahl ihrer Elemente.

Eine Sequenz s kann implizit als `bool`-Wert verwendet werden, der genau dann `true` ist, wenn die Sequenz nicht leer ist.

6.1.2 Verkettung von Sequenzen und Elementen

Für zwei Sequenzen s_1 und s_2 desselben Typs `seq<T>` liefert s_1+s_2 eine neue Sequenz mit allen Elementen von s_1 und s_2 (in dieser Reihenfolge). Wenn die Elementtypen von s_1 und s_2 verschieden sind, ist der Elementtyp der Resultatsequenz gemäß `std::common_type` definiert.

Für eine Sequenz s und einen Wert x mit einem beliebigem Typ, der nach T umwandelbar ist, ist $s+x$ bzw. $x+s$ gleichbedeutend mit $s+seq<T>(x)$ bzw. $seq<T>(x)+s$.

$s_1+=s_2$ bzw. $s+=x$ ist äquivalent zu $s_1=s_1+s_2$ bzw. $s=s+x$.

6.1.3 Positionen

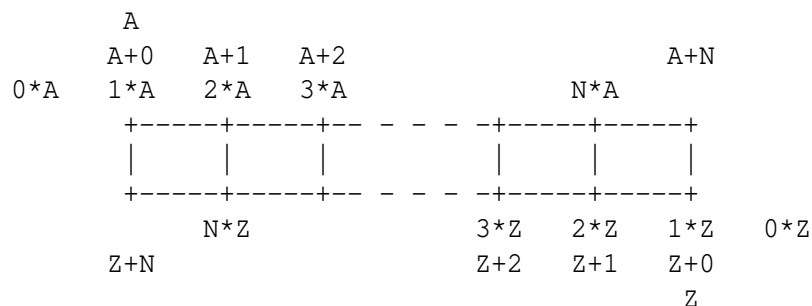
Eine Position bezeichnet logisch eine Stelle zwischen zwei Elementen (oder unmittelbar vor dem ersten bzw. nach dem letzten Element). Das bedeutet umgekehrt, dass sich ein Element immer zwischen zwei benachbarten Positionen befindet.

Die Konstante A mit Typ `posA` bezeichnet den Anfang einer beliebigen Sequenz, d. h. die Position vor dem ersten Element.

Für eine Position p mit Typ `posA` und einen ganzzahligen Wert n besitzen $p+n$ (oder $n+p$) und $p-n$ ebenfalls Typ `posA` und bezeichnen die Position, die sich n Elemente rechts bzw. links von Position p (bzw. $-n$ Elemente links bzw. rechts von p , falls n negativ ist) befindet. $n*A$ (oder $A*n$) ist gleichbedeutend mit $A+n-1$. Damit sind z. B. A , $A+0$ und $1*A$ gleichbedeutend, ebenso $A+3$ und $4*A$. Damit können, je nach persönlicher Vorliebe, Positionen entweder ab 0 oder ab 1 gezählt werden, je nachdem, ob man $A+n$ oder $n*A$ verwendet.

Vollkommen symmetrisch zu A , bezeichnet die Konstante Z mit Typ `posZ` das Ende einer Sequenz, d. h. die Position nach dem letzten Element. Entsprechend bezeichnen $p+n$ (oder $n+p$) und $p-n$ für eine Position p mit Typ `posZ` die Position, die sich n Elemente links bzw. rechts (!) von p (bzw. $-n$ Elemente rechts bzw. links von p , falls n negativ ist) befindet. $n*Z$ (oder $Z*n$) ist gleichbedeutend mit $Z+n-1$.

Die nachfolgende Abbildung zeigt die Bedeutung verschiedener Positionen für eine Sequenz mit N Elementen:



Für Positionen p und q desselben Typs liefert $q-p$ die vorzeichenbehaftete Anzahl der Elemente, die sich zwischen diesen Positionen befinden, sodass $p+(q-p)$ immer gleich q ist. Dementsprechend ist der Wert von $q-p$ für Positionen des Typs `posA` bzw. `posZ`

- positiv, wenn sich q rechts bzw. links von p befindet;
- negativ, wenn sich q links bzw. rechts von p befindet;
- null, wenn q und q dieselbe Position bezeichnen.

Für Positionen unterschiedlicher Typen ist $q-p$ jedoch nicht definiert, weil die entsprechende Anzahl der Elemente ohne Kenntnis der konkreten Sequenz nicht ermittelt werden kann.

Außerdem stehen für Positionen (ggf. desselben Typs) alle Operatoren zur Verfügung, die sich mit der „üblichen“ Bedeutung auf die oben genannten Operatoren zurückführen lassen, zum Beispiel:

- $p+=n$ entspricht $p=p+n$.
- $++p$ entspricht $p+=1$.
- $p<q$ entspricht $p-q<0$.
- usw.

Eine nicht explizit initialisierte Variable des Typs `posA` bzw. `posZ` besitzt implizit den Wert $0*A$ bzw. $0*Z$.

Eine Position p des Typs `posA` bzw. `posZ` kann implizit als `bool`-Wert verwendet werden, der genau `false` ist, wenn $p<A$ bzw. $p<Z$ ist, weil eine solche Position garantiert außerhalb jeder Sequenz (vor dem Anfang bzw. nach dem Ende) liegt.

6.1.4 Bereiche

Ein Bereich bezeichnet einen zusammenhängenden Teil einer beliebigen Sequenz, der entweder durch eine Anfangs- und eine Endposition (die unabhängig voneinander Typ `posA` oder `posZ` besitzen können) oder eine Anfangsposition und eine ganzzahlige Länge (Anzahl von Elementen) oder eine ganzzahlige Länge und eine Endposition definiert ist.

Prinzipiell sind beliebige Positionen und Längen erlaubt, das heißt:

- Jede Position kann auch vor dem Anfang oder hinter dem Ende der Sequenz liegen.
- Die Anfangsposition kann auch rechts von der Endposition liegen.
- Die Länge kann auch negativ sein.

Die Bedeutung dieser Sonderfälle wird weiter unten erklärt.

Für Positionen p und q und einen ganzzahligen Wert n bezeichnet:

- $p|q$ den Bereich mit Anfangsposition p und Endposition q ;
- $p|n$ den Bereich mit Anfangsposition p und Länge n , was äquivalent zu $p|p+n$ bzw. $p|p-n$ ist, wenn p Typ `posA` bzw. `posZ` besitzt;
- $n|q$ den Bereich mit Länge n und Endposition q was äquivalent zu $q-n|q$ bzw. $q+n|q$ ist, wenn q Typ `posA` bzw. `posZ` besitzt.

6.1.5 Bildung von Teilsequenzen

Für eine Sequenz s und einen Bereich $p|q$ mit Anfangsposition p und Endposition q liefert $s(p|q)$ eine neue Sequenz mit allen Elementen von s außer denen, die sich vor Position p oder nach Position q befinden.

Das sind „normalerweise“ die Elemente, die sich zwischen den Positionen p und q befinden. Die vorstehende Formulierung erfasst jedoch auch alle oben genannten Sonderfälle von Bereichen:

- Wenn die Anfangsposition p hinter dem Ende der Sequenz liegt, liegen alle Elemente vor dieser Position, d. h. die Teilsequenz ist dann leer.
- Ebenso, wenn die Endposition q vor dem Anfang der Sequenz liegt.
- Wenn die Anfangsposition p vor dem Anfang der Sequenz liegt, gibt es keine Elemente vor dieser Position, d. h. die Teilsequenz beginnt dann mit dem ersten Element (sofern sie nicht leer ist).
- Wenn die Endposition q nach dem Ende der Sequenz liegt, gibt es keine Elemente nach dieser Position, d. h. die Teilsequenz endet dann mit dem letzten Element (sofern sie nicht leer ist).
- Wenn die Anfangsposition p rechts von der Endposition q liegt, ist die Teilsequenz leer, weil sich dann jedes Element entweder vor Position p oder nach Position q befindet.

Für Bereiche der Art $p|n$ und $n|q$ ist $s(p|n)$ bzw. $s(n|q)$ jeweils äquivalent zu $s(p|q)$, wenn $p|q$ jeweils den zu $p|n$ bzw. $n|q$ äquivalenten Bereich gemäß der Definition weiter oben bezeichnet.

6.1.6 Ersetzung von Teilsequenzen

Für eine Sequenz s und einen Bereich $p|q$ mit Anfangsposition p und Endposition q ist $s(p|q, xs)$ äquivalent zu $s(A|p) + seq<T>(xs) + s(q|Z)$, sofern dieser Ausdruck typkorrekt ist. Dementsprechend kann xs z. B. ein einzelner Wert mit Typ T oder eine Sequenz mit Typ $seq<T>$ sein.

Das bedeutet „normalerweise“, dass in einer (gedachten) Kopie von s die Elemente zwischen den Positionen p und q durch die Elemente von xs ersetzt werden, womit auch das Einfügen und Entfernen von Elementen jeweils als Spezialfall enthalten ist. (Wenn Anfangs- und Endposition des Bereichs gleich sind, werden die Elemente von xs an dieser Position eingefügt. Wenn xs z. B. eine leere Sequenz ist, werden die Elemente des Bereichs entfernt.) Aber auch hier werden durch die vorstehende Formulierung wieder alle Sonderfälle von Bereichen berücksichtigt:

- Wenn die Anfangsposition p vor dem Anfang der Sequenz liegt, ist die Teilsequenz $s(A|p)$ leer und fehlt somit in der Resultatsequenz.
- Wenn die Endposition q nach dem Ende der Sequenz liegt, ist die Teilsequenz $s(q|Z)$ leer und fehlt somit in der Resultatsequenz.
- Wenn die Anfangsposition p rechts von der Endposition q liegt, überlappen sich die Teilsequenzen $s(A|p)$ und $s(q|Z)$ teilweise, was dazu führt, dass ihre gemeinsame Teilsequenz $s(q|p)$ in der Resultatsequenz zweimal auftritt, einmal vor und einmal nach den Elementen von xs .
Obwohl dies etwas ungewöhnlich ist, ist diese Definition zumindest konsistent und „symmetrisch“. Da der Bereich zwischen den Positionen p und q leer ist ($s(p|q)$ liefert in diesem Fall tatsächlich eine leere Sequenz), müssten die Elemente von xs eigentlich nur an der richtigen Stelle eingefügt werden. Aber die Frage nach der „richtigen“ Stelle lässt sich vermutlich nicht sinnvoll beantworten. Eine denkbare, aber noch etwas kompliziertere Alternative wäre, $s(p|q, xs)$ grundsätzlich äquivalent zu $s(A|l) + seq<T>(xs) + s(r|Z)$ zu definieren, wobei l bzw. r das „Minimum“ bzw. „Maximum“ der Positionen p und q bezeichnet.
- Wenn die Anfangsposition p hinter dem Ende der Sequenz oder die Endposition q vor dem Anfang der Sequenz liegt, enthält die Teilsequenz $s(A|p)$ bzw. $s(q|Z)$ alle Elemente von s .

Für Bereiche der Art $p|n$ und $n|q$ ist $s(p|n, xs)$ bzw. $s(n|q, xs)$ jeweils äquivalent zu $s(p|q, xs)$, wenn $p|q$ jeweils den zu $p|n$ bzw. $n|q$ äquivalenten Bereich bezeichnet.

Für eine einzelne Position p ist $s(p, xs)$ äquivalent zu $s(p|p, xs)$, d. h. die Elemente von xs werden an Position p eingefügt.

Anstelle eines einzelnen Arguments `xs` können bei allen o. g. Aufrufen auch mehrere Argumente angegeben werden, die dann jeweils einzeln mittels `seq<T>(xs)` in Sequenzen umgewandelt und dann miteinander verkettet werden.

6.1.7 Abfrage einzelner Elemente

Für eine Sequenz `s` und eine Position `p` mit Typ `posA` bzw. `posZ` liefert `s[p]` das Element von `s`, das sich rechts bzw. links von Position `p` befindet, als R-Wert. Falls es kein solches Element gibt, wird der Wert `nil` bzw. `nil_v<T>` geliefert, wobei die Variablenschablone `nil_v` wie folgt definiert ist:

```
template <typename T>
const T nil_v = T();
```

Durch Spezialisierungen dieser Schablone können bei Bedarf andere Ersatzwerte definiert werden.

6.1.8 Hilfstypen

Die Typen `boolA` und `boolZ` sind jeweils „Repliken“ von `bool`, d. h. sie besitzen jeweils implizite Umwandlungen von und nach `bool` (deren Hintereinanderausführung jeweils die identische Funktion ist) und können deshalb im wesentlichen wie `bool` verwendet werden. Ihre Unterscheidung ist jedoch im folgenden wichtig. (Deshalb ist es auch wichtig, dass es sich nicht nur um Synonyme oder Aliase von `bool` handelt, die mit `typedef` oder `using` definiert sind, sondern jeweils um eigenständige Typen.)

Die Typen `bool12A` und `bool12Z` sind ebenfalls derartige Repliken von `bool`, die zusätzlich noch einen zweiten optionalen `bool`-Wert enthalten können. Tatsächlich sind sie von `std::pair<bool, std::optional<bool>>` abgeleitet, sodass die in ihnen enthaltenen `bool`-Werte mit Hilfe der Elementvariablen `first` und `second` von `pair` und der verschiedenen Elementfunktionen von `optional` abgefragt werden können. Zusätzlich zu den impliziten Umwandlungen von und nach `bool`, die den Wert `first` setzen bzw. liefern und `second` auf `std::nullopt` setzen bzw. unbeachtet lassen, besitzen die Typen noch folgende Elementfunktionen:

- Einen Konstruktor mit zwei `bool`-Werten als Parameter, die in `first` bzw. `second` gespeichert werden.
- Einen Präfixoperator `*`, der den Wert `second` liefert.

Da `optional` ebenfalls einen Präfixoperator `*` sowie eine implizite Umwandlung nach `bool` besitzt, können die in einem Objekt `x` des Typs `bool12A` oder `bool12Z` enthaltenen `bool`-Werte auch wie folgt abgefragt werden:

- Die implizite Verwendung von `x` als `bool`-Wert liefert den ersten `bool`-Wert `x.first`.
- Die implizite Verwendung von `*x` als `bool`-Wert liefert den Wert `x.second.has_value()`, der genau dann `true` ist, wenn `x` einen zweiten `bool`-Wert enthält.
- In diesem Fall liefert `**x` diesen zweiten `bool`-Wert `x.second.value()`. (Andernfalls ist `**x` undefiniert.)

Obwohl auch diese beiden Typen „funktional identisch“ sind, ist ihre Unterscheidung im folgenden wichtig.

6.1.9 Suchen, Filtern und Transformieren von Elementen

Für eine Sequenz `s` und eine Funktion `f` (bei der er sich auch um ein Funktionsobjekt oder einen Lambda-Ausdruck handeln kann), die mit einem Wert des Typs `T` aufgerufen werden kann und ein Resultat eines beliebigen Typs `R` zurückliefert, liefert `s(f)` entweder einen Wert des Typs `posA` oder `posZ` oder eine neue Sequenz, deren Elemente wie folgt aus den Elementen `x` von `s` gebildet werden:

- Wenn R gleich `boolA` oder `boolZ` ist, hat der Resultatwert von $s(f)$ den Typ `posA` bzw. `posZ` und bezeichnet „normalerweise“ die Position p des ersten Elements von vorn bzw. von hinten, für das $f(s[p])$ gleich `true` ist. Falls es kein solches Element gibt, ist der Resultatwert `0*A` bzw. `0*Z`. (Das heißt, der Resultatwert ist genau implizit `true`, wenn es ein entsprechendes Element gibt.) Das heißt, die Funktion f wird zum Suchen des ersten bzw. letzten Elements mit einer bestimmten Eigenschaft verwendet.
- Wenn R gleich `bool` ist, besitzt die Resultatsequenz den Typ `seq<T>` und enthält diejenigen Elemente x , für die $f(x)$ gleich `true` ist. Das heißt, die Funktion f wird zum Filtern der Elemente x verwendet. Dieses Verhalten kann prinzipiell genauso leicht auch mit dem nachfolgenden Fall erreicht werden. Die Verwendung von `bool` anstelle von `bool12A` oder `bool12Z` als Resultattyp verdeutlicht jedoch zum einen, dass die Reihenfolge, in der die Elemente durchlaufen werden, hier prinzipiell unwichtig ist; zum anderen lässt sich dieser Fall etwas effizienter implementieren, weil die Resultatwerte der Funktionsaufrufe $f(x)$ nur auf `true` oder `false` getestet und nicht genauer untersucht werden müssen.
- Wenn R gleich `bool12A` oder `bool12Z` ist, besitzt die Resultatsequenz ebenfalls den Typ `seq<T>` und enthält zunächst diejenigen Elemente x , für die $f(x).first$ gleich `true` ist. Sobald $f(x).second.has_value()$ bei einem Aufruf von f (unabhängig vom Wert $f(x).first$) gleich `true` ist, wird die Funktion für die verbleibenden Elemente x nicht mehr aufgerufen; wenn $f(x).second.value()$ dann ebenfalls `true` ist, werden diese verbleibenden Elemente noch zur Resultatsequenz hinzugefügt, andernfalls nicht. Wenn R gleich `bool12A` ist, werden die Elemente x hierfür in ihrer „natürlichen“ Reihenfolge von vorn nach hinten durchlaufen. Wenn R gleich `bool12Z` ist, werden sie jedoch in umgekehrter Reihenfolge durchlaufen und die Resultatsequenz entsprechend von hinten nach vorn aufgebaut. Dieser Unterschied im Resultattyp der Funktion ist genau dann wichtig, wenn der Fall $f(x).second.has_value()$ eintritt. (Wenn bekannt ist, dass er nie eintreten wird, kann statt `bool12A` oder `bool12Z` einfach Resultattyp `bool` verwendet werden, wie weiter oben bereits erläutert wurde.) Das heißt, die Funktion f wird hier ebenfalls zum Filtern einzelner Elemente verwendet und bietet gleichzeitig die Möglichkeit, alle verbleibenden Elemente ab einer bestimmten Stelle auf einmal ein- oder auszuschließen. Damit lassen sich insbesondere Funktionen zum bequemen Löschen des ersten (Resultattyp `bool12A`) oder letzten (Resultattyp `bool12Z`) Elements mit einer bestimmten Eigenschaft schreiben (siehe unten).
- Wenn R gleich `std::optional<U>` für einen beliebigen Typ U ist, besitzt die Resultatsequenz den Typ `seq<U>` und enthält alle Elemente $f(x).value()$, für die $f(x).has_value()$ gleich `true` ist. Das heißt, die Funktion f wird hier sowohl zum Filtern als auch zum Transformieren von Elementen verwendet.
- Für alle anderen Typen R besitzt die Resultatsequenz den Typ `seq<R>` und enthält alle Elemente $f(x)$. Das heißt, die Funktion f wird hier nur zum Transformieren von Elementen verwendet. Wenn eine solche Transformationsfunktion als Resultattyp R eigentlich einen der zuvor genannten besonderen Typen (z. B. `bool` oder `optional<U>`) besitzen soll, kann man stattdessen Resultattyp `optional<R>` (also z. B. `optional<bool>` oder `optional<optional<U>>`) verwenden und immer ein `optional`-Objekt zurückliefern, das tatsächlich einen Wert enthält.

Neben $s(f)$ ist auch die allgemeinere Form $s(f, args)$ mit beliebigen weiteren Argumenten $args$ möglich, die dann jeweils zusätzlich zum aktuellen Element x an die Funktion f weitergegeben werden.

6.1.10 Funktionen zum Suchen und Löschen bestimmter Elemente

Für eine Funktion f mit Resultattyp `bool` (z. B. `eq(5)` oder `(rem(2), eq(0))`) liefert:

- `s(searchA(f, n))` bzw. `s(searchZ(f, n))` die Position mit Typ `posA` bzw. `posZ` des n -ten Elements x von s von vorn bzw. von hinten, für das $f(x)$ gleich `true` ist. Wenn es kein solches Element gibt, ist der Resultatwert `0*A` bzw. `0*Z`.
Wenn das Argument n fehlt, wird $n=1$ verwendet.
- `s(remove(f))` eine Kopie von s , in der alle Elemente x fehlen, für die $f(x)$ gleich `true` ist.
- `s(removeA(f, n))` bzw. `s(removeZ(f, n))` eine Kopie von s , in der die ersten bzw. letzten n Elemente x fehlen, für die $f(x)$ gleich `true` ist.
Wenn das Argument n fehlt, wird $n=1$ verwendet.

6.1.11 STL-Iteratoren

Für eine Sequenz s und eine Position p mit Typ `posA` bzw. `posZ` liefert `s(p)` einen konstanten STL-Iterator der Kategorie Random Access, der logisch auf die Position p von s zeigt (die sich zwischen zwei benachbarten Elementen befindet). Aus STL-Sicht zeigt er jedoch immer auf das Element `s[p]`, egal ob es dieses Element tatsächlich gibt oder nicht, sodass `*s(p)` immer äquivalent zu `s[p]` ist. Des Weiteren ist auch `*(s(p)+n)` für jeden ganzzahligen Wert n äquivalent zu `s[p+n]`. Daraus folgt indirekt, dass die von `s(p)` gelieferten Iteratoren für p vom Typ `posA` bzw. `posZ` vorwärts bzw. rückwärts (reverse) durch die Sequenz s iterieren.

Für einen derartigen Iterator p liefert `~p` einen korrespondierenden Reverse-Iterator, der logisch auf dieselbe Position von s wie p zeigt. Aus STL-Sicht zeigt `~p` jedoch auf das Element auf der jeweils anderen Seite dieser Position.

Insbesondere gilt (logisch, da es die entsprechenden Elementfunktionen nicht gibt):

- `s(A)` entspricht `s.cbegin()`.
- `s(Z)` entspricht `s.crbegin()`.
- `~s(Z)` entspricht `s.cend()`.
- `~s(A)` entspricht `s.crend()`.

Darüber hinaus gibt es die für die Anweisung `for (x : s)` benötigten Elementfunktionen `begin` und `end` (die ebenfalls konstante Iteratoren liefern).

6.1.12 Vergleichsoperatoren

Für zwei Sequenzen $s1$ und $s2$ desselben Typs `seq<T>` liefert `diff(s1, s2)` das Ergebnis des „Dreiwegvergleichs“ von $s1$ und $s2$, d. h. einen ganzzahligen Wert kleiner bzw. gleich bzw. größer als 0, wenn $s1$ lexikographisch kleiner bzw. gleich bzw. größer als $s2$ ist.

Zwei Werte $x1$ und $x2$ des Typs T werden dabei wiederum mittels `diff(x1, x2)` verglichen. Hierfür ist `diff` als Funktionsschablone für beliebige Typen T vordefiniert und liefert jeweils die Differenz $x1-x2$. Durch Spezialisierungen dieser Schablone kann dieser „Dreiwegvergleich“ für bestimmte Typen T aber auch anders definiert werden.

Für zwei Sequenzen $s1$ und $s2$ desselben Typs `seq<T>` und einen Vergleichsoperator `op` aus der Menge `<, >, <=, >=, ==, !=` ist `s1 op s2` gleichbedeutend mit `diff(s1, s2) op 0`.

6.1.13 Zeichenketten

`str` ist ein Synonym bzw. Alias für `seq<char>`.

Dieser Typ besitzt einen zusätzlichen impliziten Konstruktor mit Parametertyp `const char*`, sodass „C-Strings“ (insbesondere String-Literale) bei Bedarf implizit in `str` umgewandelt werden und deshalb (vermutlich) überall verwendet werden können, wo ein Wert des Typs `str` erwartet wird, z. B. bei Verkettungsoperationen oder Vergleichen.

Außerdem ist `std::hash<str>` sowie der übliche Ausgabeoperator `<<` für `str` definiert.

6.1.14 Sonstiges

Für eine Sequenz `s` liefert `~s` eine neue Sequenz, die die Elemente von `s` in umgekehrter Reihenfolge enthält.

`is_seq` und `is_seq_v` sind analog zu `std::is_pointer` und `std::is_pointer_v` und ähnlichen Schablonen definiert und können verwendet werden, um zur Übersetzungszeit zu überprüfen, ob ein Typ ein Sequenztyp ist.

7 Offene Typen (`open.ch`)

7.1 Beschreibung

7.1.1 Offene Typen

`TYPE(X)` definiert einen offenen Typ mit dem Namen `X` mit folgenden Konstruktoren und weiteren Operationen:

- `X()` liefert ein `nil`-Objekt, d. h. logisch kein Objekt.
Dieser Konstruktor kann auch implizit aufgerufen werden.
- `X(uniq)` liefert ein neues leeres Objekt mit einer eindeutigen Identität. (`uniq` ist eine globale Konstante.)
Dieser Konstruktor kann auch implizit aufgerufen werden.
- Für ein Attribut `a` von `X` (siehe unten) und prinzipiell beliebige Werte `y` ist `X(a, y ...)` äquivalent zu `X(uniq)(a, y ...)`.
Die Bedeutung des hier verwendeten Klammeroperators wird später erklärt.
- Für ein Objekt `x` des Typs `X` liefert der Kopierkonstruktor `X(x)` ein Objekt mit derselben Identität wie `x`, d. h. logisch dasselbe Objekt.
Dieser Konstruktor kann auch implizit aufgerufen werden.
- Für Objekte `x1` und `x2` des Typs `X` überschreibt die Zuweisung `x1=x2` die Identität von `x1` durch die Identität von `x2`, d. h. anschließend enthält die Variable `x1` logisch dasselbe Objekt wie `x2`.
- Ein Objekt `x` des Typs `X` kann implizit als `bool`-Wert verwendet werden, der genau dann `true` ist, wenn `x` ein „echtes“ Objekt ist, das direkt oder indirekt mittels `X(uniq)` initialisiert wurde.
- Für Objekte `x1` und `x2` des Typs `X` kann mittels `x1==x2` und `x1!=x2` überprüft werden, ob sie dieselbe Identität besitzen oder nicht.

Durch eine eigene Definition von `operator==` für den Typ `X` (als normale Funktion oder Funktionsschablone) kann ein beliebiges anderes Vergleichskriterium definiert werden. Der vordefinierte

operator!= ruft lediglich den entsprechenden operator== auf, sodass er nicht separat undefiniert werden muss.

- Für ein „echtes“ Objekt x eines offenen Typs X liefert $+x$ ein Duplikat des Objekts, d. h. ein neues Objekt des Typs X mit denselben Attributwerten wie x . (Das heißt, sämtliche Attributwerte von x werden mit ihren jeweiligen Kopierkonstruktoren oder Zuweisungsoperatoren in das neue Objekt kopiert.) Für ein nil-Objekt x liefert $+x$ ebenfalls ein nil-Objekt.

7.1.2 Attribute

Für einen Namen a , einen offenen Typ X und einen (nahezu) beliebigen Typ Y definiert $ATTR1(a, X, Y)$ bzw. $ATTRN(a, X, Y)$ ein einwertiges bzw. mehrwertiges Attribut a des Typs X mit Zieltyp Y .

Unterschiedliche Typen können Attribute mit demselben Namen und gleichen oder unterschiedlichen Zieltypen besitzen.

7.1.3 Einwertige Attribute

Für einen offenen Typ X , ein Objekt x dieses Typs und ein einwertiges Attribut a dieses Typs mit Zieltyp Y liefert $x(a)$ den aktuellen Wert des Attributs a des Objekts x als R-Wert. Falls das Attribut für dieses Objekt noch keinen Wert besitzt, erhält man den Ersatzwert `nil`.

Für einen Wert y mit Typ Y setzt $x(a, y)$ den Wert des Attributs a des Objekts x auf y und liefert das Objekt x zurück. Damit können beliebig viele derartige Operationen bequem nacheinander ausgeführt werden:

$$x(a1, y1) (a2, y2) \dots\dots$$

Durch Kombination mit dem Konstruktor des Typs X können außerdem bequem Objekte mit beliebig vielen initialen Attributwerten erzeugt werden:

$$X(a1, y1) (a2, y2) \dots\dots$$

Wenn x ein nil-Objekt ist, ist $x(a, y)$ wirkungslos.

7.1.4 Mehrwertige Attribute

Ein mehrwertiges Attribut a mit Zieltyp Y entspricht zunächst einem einwertigen Attribut mit Zieltyp $seq<Y>$, d. h. $x(a)$ liefert die aktuellen Werte des Attributs a des Objekts x als (ggf. leere) Sequenz mit Typ $seq<Y>$ und $x(a, ys)$ ersetzt diese Werte durch die Elemente der Sequenz ys .

Neben dieser grundlegenden „Schreibfunktion“ zum Ersetzen aller Werte auf einmal, gibt es jedoch weitere Schreibfunktionen $x(a, y \dots)$ mit folgender Bedeutung:

- Wenn der erste Parameter y (nach dem Attribut a) ein einzelner Wert mit Typ Y oder eine Sequenz mit Typ $seq<Y>$ ist, werden die aktuellen Werte des Attributs a des Objekts x durch die Werte aller Parameter y ersetzt, sofern deren Verkettung zu einer einzigen Sequenz typkorrekt ist. Dies schließt die zuvor beschriebene grundlegende Schreibfunktion als Spezialfall mit ein.
- Andernfalls (d. h. wenn der erste Parameter y weder in Y noch in $seq<Y>$ umgewandelt werden kann) werden alle aktuellen Werte des Attributs durch die Werte der Sequenz $x(a) (y \dots)$ ersetzt, d. h. alle Parameter y werden an den Klammeroperator der aktuellen Sequenz $x(a)$ übergeben, der je nach Art der Parameter z. B. eine Teilsequenz davon liefert oder eine Kopie der Sequenz, in der bestimmte Elemente durch andere ersetzt sind, usw. (Vgl. die Beschreibung von Sequenzen.)

Wenn x ein nil-Objekt ist, ist $x(a, y \dots)$ wiederum wirkungslos.

Anmerkung: In der ersten Version der Bibliothek vom März 2021 hatte der erste dieser zwei Fälle eine andere Bedeutung: Die Werte aller Parameter y wurden zu den bereits vorhandenen Werten hinzugefügt anstatt diese zu ersetzen. Obwohl dies den Vorteil hatte, dass das Hinzufügen von Werten damit maximal einfach und bequem möglich war, hatte es den entscheidenden Nachteil, dass die Operationen $x(a1, ys)$ und $x(a2, ys)$ für ein einwertiges Attribut $a1$ mit Zieltyp $seq<Y>$ und ein mehrwertiges Attribut $a2$ mit Zieltyp Y unterschiedliche Bedeutung hatten, obwohl die Attribute $a1$ und $a2$ logisch dasselbe darstellen: $x(a1, ys)$ ersetzt alle aktuellen Attributwerte durch die Werte der Sequenz ys , während $x(a2, ys)$ die Werte der Sequenz hinzufügt. Mit der jetzt gültigen Definition bewirken beide Operationen dasselbe, nämlich eine Ersetzung der aktuellen Attributwerte.

7.1.5 Typ- und Attributschablonen

`TYPETEMP(X, tpar ...)` definiert eine Schablone (template) von offenen Typen mit dem Namen X und einem oder mehreren Schablonenparametern $tparam$, zum Beispiel (die Namen der Schablonenparameter können auch weggelassen werden):

```
TYPETEMP(List, typename T)
TYPETEMP(Pair, typename U, typename V)
```

`ATTR1TEMP(a, X, Y, tpar ...)` bzw. `ATTRNTEMP(a, X, Y, tpar ...)` definiert eine Schablone von ein- bzw. mehrwertigen Attributen mit dem Namen a und Schablonenparametern $tparam$ für die Typen X mit Zieltyp(en) Y . X muss ein Schablonenbezeichner (template id) sein, aus dem alle Schablonenparameter deduziert werden können, z. B. `List<T>` oder `Pair<U, V>`. Y kann ein normaler Typbezeichner oder ebenfalls ein Schablonenbezeichner sein, der beliebig von einem oder mehreren Schablonenparametern abhängen kann (d. h. die Parameter müssen aus Y nicht deduziert werden können).

Wenn X oder Y Kommas enthalten, müssen sie mit `LIT` (definiert in `util.ch`) geklammert werden, damit sie jeweils als ein einziges Makroargument übergeben werden, zum Beispiel:

```
ATTR1TEMP(head, List<T>, T, typename T)
ATTR1TEMP(first, LIT(Pair<U, V>), U, typename U, typename V)
```

7.1.6 Definitionen in Namensräumen und anderen Dateien

Wenn Typen und Attribute in mehreren Quelldateien benötigt werden, können sie – analog zu Strukturen bzw. Klassen und inline-Funktionen – in Definitionsdateien stehen, die dann in mehrere Quelldateien eingebunden werden können.

Typen und Attribute können prinzipiell unabhängig voneinander in unterschiedlichen Namensräumen definiert werden. Insbesondere kann in einem Namensraum ein Attribut für einen Typ definiert werden, der in einem anderen Namensraum definiert wurde. Dieses Attribut kann sogar den gleichen Namen (und den gleichen oder einen anderen Zieltyp) besitzen wie ein anderes Attribut des Typs, das in einem anderen Namensraum definiert wurde.

Die Namen der Typen und Attribute müssen hierfür bei Bedarf nach den normalen Regeln von C++ qualifiziert oder mittels `using` sichtbar gemacht werden.

7.1.7 Virtuelle Attribute

Virtuelle Attribute können wie gewöhnliche Attribute verwendet werden, obwohl ihre Werte nicht direkt in Objekten offener Typen gespeichert werden, sondern von benutzerdefinierten Funktionen nach Belieben gelesen und gespeichert werden können.

Ein virtuelles Attribut mit dem Namen `a` muss zunächst (unabhängig von einem konkreten Typ) mittels `ATTR(a)` definiert werden, sofern nicht bereits ein Attribut mit dem gleichen Namen mittels `ATTR1`, `ATTRN`, `ATTR1TEMP` oder `ATTRNTEMP` (im selben Namensraum in derselben Datei) definiert wurde. (Wenn bereits ein gleichnamiges Attribut definiert wurde, ist die Definition `ATTR(a)` nicht nötig, aber auch nicht störend.)

Anschließend können beliebig viele zugehörige Funktionen definiert werden, die alle den „virtuellen“ Namen `FUNC` besitzen (in Wirklichkeit ist `FUNC` bzw. `CH_FUNC` ein Makro) und deren Parameterliste mit dem Attributnamen `a` beginnt (obwohl `a` in Wirklichkeit kein Typname ist).

Der zweite Parameter muss ein Objekt `x` des offenen Typs `X` sein, zu dem das Attribut gehören soll. Eventuelle weitere Parameter `y` ergeben sich aus den gewünschten Aufrufmöglichkeiten, weil jeder Aufruf der Art `x(a, y ...)` letztlich auf einen Aufruf `FUNC(aa, x, y ...)` einer so definierten Funktion abgebildet wird. Dabei steht `aa` für irgendeinen geeigneten Wert des ersten Parameters.

Zum Beispiel:

```
// Offener Typ Person mit mehrwertigem Attribut firstnames.
TYPE(Person)
ATTRN(firstnames, Person, str)

// Virtuelles Attribut firstname für prinzipiell beliebige offene Typen
// mit prinzipiell beliebigen Zieltypen.
ATTR(firstname)

// Einwertiges virtuelles Attribut firstname für Person mit Zieltyp
// str durch eine Lese- und eine Schreibfunktion definieren.
// Die Lesefunktion liefert den ersten Vornamen der Person p, falls
// vorhanden, andernfalls nil.
// Die Schreibfunktion ersetzt den ersten Vornamen der Person p durch s,
// falls es bereits einen gibt, andernfalls wird s als erster Vorname
// hinzugefügt.
// Die Schreibfunktion sollte das Objekt p zurückliefern, damit
// verkettete Verwendungen des Klammeroperators wie gewohnt
// funktionieren.
str FUNC (firstname, Person p) {
    return p(firstnames)[A];
}
Person FUNC (firstname, Person p, const str& s) {
    return p(firstnames, A|1, s);
}

// Verwendung von firstname wie ein anderes einwertiges Attribut,
// zum Beispiel:
Person p1(firstname, "Hans");
str s1 = p1(firstname);
```

7.1.8 Streuwertfunktionen für offene Typen

Für jeden offenen Typ `X` ist der Typ `hash<X>` analog zu `std::hash` definiert, d. h. er besitzt einen konstanten Klammeroperator mit Parametertyp `T` und Resultattyp `size_t`, der für ein Objekt des Typs `T` seinen Streuwert anhand seiner Identität berechnet und zurückliefert.

Damit kann `std::hash<X>` einfach wie folgt definiert werden:

```
template <>
struct std::hash<X> : CH::hash<X> {};
```

Weil diese Definition aber nur im globalen Namensraum erlaubt ist, kann sie im allgemeinen nicht durch `TYPE(X)` bereitgestellt werden. Außerdem besteht so die Möglichkeit, `std::hash<X>` für einen Typ bei Bedarf auch anders zu definieren.

7.1.9 Sonstiges

Für einen Attributnamen `a` liefert `a()` ein generisches Funktionsobjekt, sodass jeder Aufruf der Art `a(x, ...)` äquivalent zum Aufruf `x(a, ...)` ist.

Derartige Funktionsobjekte können u. a. zum Transformieren von Sequenzelementen verwendet werden (vgl. §6.1.9). Wenn `ps` beispielsweise eine Sequenz von Personen `p` ist und der Typ `Person` einwertige (gewöhnliche oder auch virtuelle) Attribute `name` und `spouse` besitzt, liefert `ps(spouse())` eine Sequenz mit den Ehepartnern `p(spouse)` und `ps(spouse())(name())` eine Sequenz mit den Namen `p(spouse)(name)` der Ehepartner der Personen `p`. (Falls eine Person keinen Ehepartner besitzt, ist der entsprechende Name `nil`, also eine leere Zeichenkette.)

`is_open` und `is_open_v` sind analog zu `std::is_pointer` und `std::is_pointer_v` und ähnlichen Schablonen definiert und können verwendet werden, um zur Übersetzungszeit zu überprüfen, ob ein Typ ein offener Typ ist.

7.1.10 Anmerkungen

Offene Typen haben gegenüber gewöhnlichen Strukturen oder Klassen einige Vorteile:

Zum einen können Attribute auch noch „später“ oder an einer anderen Stelle des Programms zu einem Typ hinzugefügt werden. So kann jedes „Modul“ bei Bedarf weitere Attribute definieren, von denen die übrigen Module u. U. gar nichts wissen müssen.

Zum anderen muss ein bestimmtes Objekt eines offenen Typs nicht immer Werte für sämtliche Attribute des Typs besitzen. Nicht vorhandene Attributwerte belegen dann auch keinen Speicherplatz. Beim Zugriff auf den Wert eines nicht vorhandenen Attributs erhält man einen wohldefinierten `nil`-Wert.

Damit decken offene Typen auch bequem variante Strukturen ab, die man ansonsten relativ aufwendig entweder mit `union` oder mit `std::variant` oder mit „künstlichen“ Klassenhierarchien (z. B. eine abstrakte Basisklasse `Expr` für arithmetische Ausdrücke mit konkreten Unterklassen wie `Const` für konstante Ausdrücke und `Add`, `Sub` etc. für Addition, Subtraktion etc.) modellieren müsste. Anders als bei Lösungen mit `union`, braucht man normalerweise auch kein explizites „tag“, um die konkret vorliegende Variante zu erkennen, sondern kann einfach die charakteristischen Attribute der einzelnen Varianten abfragen und damit indirekt die vorliegende Variante erkennen.

Änderungsprotokoll

2021-03-22

- Erste Version.

2021-03-24

- Datei `aux.ch` in `util.ch` umbenannt, weil Windows den Namen `aux` nicht mag.
- Makro `CH_IF` verallgemeinert.
- Den variadischen Konstruktor von `seq` ein wenig umgeschrieben, damit er von `msvc` akzeptiert wird.

2021-04-03

- Kleine Fehler in der Beschreibung offener Typen korrigiert (`EMPTYTYPE` → `TYPETEMP` etc.).

2021-04-27

- Bedeutung von `x(a, ys)` für mehrwertige Attribute `a` geändert, damit man sie mit der Bedeutung für einwertige Attribute übereinstimmt.
- Konstruktoren offener Typen geändert.
- Typ `bool3` mit zugehörigen Operatoren zu `util.ch` hinzugefügt.